DOCUMENT RESUME

ED 281 208 CS 210 427

AUTHOR Mayer, John; Kieras, David E.
TITLE A Development System for Augmented Transition Network Grammars and a Large Grammar for Technical Prose. Technical Report No. 25.
INSTITUTION Michigan Univ., Ann Arbor.
SPONS AGENCY Office of Naval Research, Arlington, Va. Personnel and Training Research Programs Office.
REPORT NO ONR-TR-87-25
PUB DATE 15 Mar 87
CONTRACT N00014-85-K-0385
NOTE 41p.
PUB TYPE Viewpoints (120) -- Information Analyses (070)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS Algorithms; *Authoring Aids (Programing); Computer Networks; *Computer Oriented Programs; Grammar; Language Processing; *Programing Languages; Reader Text Relationship; *Resource Materials; Semantics; Syntax; *Technical Writing; *Training Methods
IDENTIFIERS *Augmented Transition Network Grammars; Parsing

ABSTRACT

Using a system based on standard augmented transition network (ATN) parsing approach, this report describes a technique for the rapid development of natural language parsing, called High-Level Grammar Specification Language (HGSL). The first part of the report describes the syntax and semantics of HGSL and the network implementation of each of its constructs, while the second section discusses the algorithms used in the HGSL compiler and the ATN interpreter. The third section presents a large grammar for technical prose that was developed with the system and which allows parsing of technical training materials in the draft stage of writing as part of a computer-based comprehensible writing aid. The report concludes with a review of some of the results on the coverage of the grammar. The grammar for technical training materials is appended. (FL)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)
This document has been reproduced as received from the person or organization originating it.
☐ Minor changes have been made to improve reproduction quality.
• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

ED282208

A Development System for Augmented Transition Network Grammars and A Large Grammar for Technical Prose

John Mayer and David Kieras

University of Michigan

Technical Report No. 25 (TR-87/ONR-25)

March 15, 1987

This research was supported by the Personnel and Training Research Programs under Contract Number N00014-85-K-0385, Contract Authority Identification Number NR 667-547. Reproduction in whole or part is permitted for any purpose of the United States Government.

Approved for Public Release; Distribution Unlimited

CS210427

BEST COPY AVAILABLE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a REPORT SECURITY CLASSIFICATION: Unclassified	1b RESTRICTIVE MARKINGS:
2a SECURITY CLASSIFICATION AUTHORITY:	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release: distribution unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE:	
4 PERFORMING ORGANIZATION REPORT NUMBER(S): TR-87/ONR-25	5 MONITORING ORGANIZATION REPORT NUMBER(S):
6a NAME OF PERFORMING ORGANIZATION: University of Michigan; 6b OFFICE SYMBOL (If applicable):	7a NAME OF MONITORING ORGANIZATION: Cognitive Science Office of Naval Research (Code 1142CS) 800 N. Quincy Street
6c ADDRESS (City, State, and ZIP Code): Technical Communication Program Ann Arbor, MI 48109-1109	7b ADDRESS (City, State, and ZIP Code): Arlington, VA 22217
8a NAME OF FUNDING/SPONSORING ORGANIZATION: ; 8b OFFICE SYMBOL (If applicable):	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-85-K-0385
8c ADDRESS (City, State, and ZIP Code):	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO: 61153N; PROJECT NO: RR04206; TASK NO: RR04206-0A; WORK UNIT ACCESSION NO: NR667-547

11 TITLE (Include Security Classification): A Development System for Augmented Transition Network Grammars and a Large Grammar for Technical Prose

12 PERSONAL AUTHOR(S): John Mayer and David E. Kieras

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Technical	FROM _____ TO _____	March 15, 1987	48

16 SUPPLEMENTARY NOTATION:

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Training Materials, Documentation, Authoring Systems, Natural Language Processing
05	09		

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This report is in two major sections. The first presents a High-Level Grammar Specification Language (HGSL) which greatly simplifies the development of a complex augmented transition network grammar (ATN). A compiler converts HGSL expressions into a transition network which a simple interpreter uses for parsing. The algorithms used by the compiler and interpreter are presented. The second section presents the HGSL for a large grammar for technical prose. The grammar was developed to allow parsing of technical training materials in the draft stage of writing, as part of a computer-based comprehensible writing aid. Some results on the coverage of the grammar are presented to show that the grammar is close to being practically useful.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	
22a NAME OF RESPONSIBLE INDIVIDUAL: Susan Chipman	22b TELEPHONE (Include Area Code): (202) 696-4318; 22c OFFICE SYMBOL:

DD Form 1473, JUN 86 *Previous editions are obsolete*

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

ABSTRACT

This report is in two major sections. The first presents a High-Level Grammar Specification Language (HGSL) which greatly simplifies the development of a complex augmented transition network grammar (ATN). A compiler converts HGSL expressions into a transition network which a simple interpreter uses for parsing. The algorithms used by the compiler and interpreter are presented. The second section presents the HGSL for a large grammar for technical prose. The grammar was developed to allow parsing of technical training materials in the draft stage of writing, as part of a computer-based comprehensible writing aid. Some results on the coverage of the grammar are presented to show that the grammar is close to being practically useful.

A Development System for Augmented Transition Network Grammars and a Large Grammar for Technical Prose

John Mayer and David Kieras

The system described in this report is meant to allow for the rapid development of augmented transition network (ATN) parsers for natural language parsing. This report assumes knowledge of the basics of ATN parsers; for background, see Winograd (1983). The system is based on the standard ATN parser approach, but the user does not directly specify the nodes and arcs of the network grammar to be interpreted; rather, the grammar developer uses a more abstract shorthand called High-level Grammar Specification Language (HGSL). An HGSL compiler converts this shorthand into equivalent networks suitable for use by the ATN interpreter.

The first part of this report describes the syntax and semantics of HGSL, and the network implementation of each construct. The second part describes the algorithms used in the HGSL compiler and the ATN interpreter and HGSL compiler. The third part presents a large grammar for technical prose which was developed with this system.

A High-level Grammar Specification Language (HGSL)

HGSL allows the user to easily specify common syntactic patterns. These specifications are then compiled into ATN networks, which are interpreted during parsing. These ATN networks are constructed from arcs of five types: word test, lex test, net-call, pop, and conditional. Word test arcs allow control to pass to the next node only if the current word matches the given word. Lex test arcs are similar but specify a lexical category to match, rather than a particular word. Net-call arcs name some network which is to be called, together with a next state to which control passes if the net call succeeds, while a pop arc signals a successful return from a net call. Last, the conditional arc causes evaluation of an arbitrary condition, which if true, results in control passing to the specified next state.

The language allows for matching the input sentence against actual words, lexical categories, networks, and conditions. These basic components may be combined to form sequences and alternations. Optional and repeated items are indicated in a straightforward way.

HGSL Syntax

In the following discussion, we will present the syntax of each HGSL construct, its meaning, an example of its use, and finally the ATN network into which it is compiled. Table 1 gives a context-free grammar for HGSL; rules from this grammar will be cited for each construct.

Table 1

Context-free Grammar for the High-level Specification Language

1	Grammar	->	Netdefinition Grammar
2		->	Netdefinition "END-GRAMMAR"
3	Netdefinition	->	"NET-DEF" <string> Expression
4	Expression	->	"!"<string>
5	Expression	->	<string>
6	Expression	->	"#"<string>
7	Expression	->	"{" Expression Sequence
8	Sequence	->	Expression Sequence
9		->	"}"
10	Expression	->	"{" Expression Alternation
11	Alternation	->	"/" Expression Alternation
12		->	"}"
13	Expression	->	"-" "{" Expression "}"
14	Expression	->	"+" "{" Expression "}"
15	Expression	->	"*" "{" Expression "}"
16	Expression	->	"<" <lisp expression> ">"

A grammar written in HGSL is a list of network definitions, each definition consisting of the key word NET-DEF, followed by the name of the network and an HGSL expression. The list is terminated with the key word END-GRAMMAR (Rules 1-3). The top-level network must be named #START.

Basic expressions. The simplest HGSL expressions are used to match actual words or lexical categories. A literal word match is specified by prefixing the relevant word with an exclamation point (Rule 4). Thus the expression !THE will recognize only the word the. The network which is built to implement a literal word match is a single word-test arc that compares the current word with <string>. Far more useful is the ability to specify a lexical category match. Since this is the most common test in a grammar, lexical categories are written plainly (Rule 5). For example, the expression NOUN will match any noun. The lexical category match generates a single lex-test

arc. A network match is indicated by prefixing a pound sign to the name of the network (Rule 6). For example, #NP is an invocation of the noun phrase network. The net-call expression generates a net-call arc.

Sequences. A sequence can be described by enclosing a list of HGSL expressions in brackets (Rules 7-9). Thus {!THE NOUN #VP} is a sequential pattern satisfied by the word the, followed by any word of the class noun, followed by any group of words which satisfies the expression for the #VP network. This rule can be applied recursively, allowing us to create a sequential expression from simpler sequential expressions. For example, {#NP {#VP #NP}} is a legal expression which happens to be equivalent to {#NP #VP #NP}.

Alternations. To match exactly one of several expressions, the alternatives are separated by slashes and the whole is enclosed in brackets (Rules 10-12). The pattern {!THE / !A / !SOME} requires the next word to be one of the three words, the, a, or some. Once we have both sequences and alternations, the recursive possibilities of HGSL become more interesting as in {#VP / {#NP #VP}}. This pattern could be a top-level definition of #SENTENCE since it is satisfied either by #VP (an imperative sentence), or by the sequence {#NP #VP} (a declarative pattern).

Optional matches. The appearance of a subexpression in some larger pattern may be made optional by placing a dash before it (Rule 13). The pattern { - {!IN !ORDER} !TO #VP} matches both In order to form a more perfect union and To form a more perfect union.

Repetition. Shorthand expressions are provided for two very common types of sequential repetition corresponding to zero-or-more, shown by a preceding asterisk, and one-or-more shown by a preceding plus sign (Rules 14, 15). For example, {* {PREP #NP}} matches any number of consecutive prepositional phrases, including none at all, and {#NP !VERB + {#NP}} matches a sentence with one or more objects in the verb phrase.

Conditional matches. It will sometimes be convenient to be able to insert arbitrary conditions into a larger expression. A LISP form that evaluates to true or false can be enclosed in angle brackets to constitute a valid HGSL expression. For example, the pattern

{#NP #VP <EQUAL (NUMBER-OF NP) (NUMBER-OF VP) > #NP}

first matches an #NP followed by a #VP. We must then evaluate the condition in angle brackets and then proceed to match a second NP only if the condition evaluates to true. The condition is implemented by a single test arc in the ATN. Note that HGSL does

not provide any standardized data structures to be tested by conditional expressions. Thus in order to write a condition, the grammar writer must go outside HGSL, at least in its current form, and devise a LISP expression based on the data structures of the interpreter.

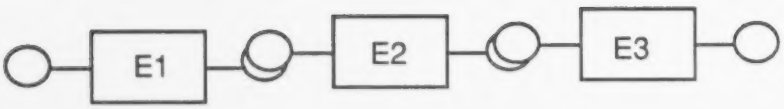
These conditions may be arbitrarily complicated and therefore may be a trap for the grammar developer. Using them too often will severely reduce the ease with which the grammar can be understood and extended. On the other hand, a few well-motivated conditions may allow considerable rule economy without introducing any serious obscurity. Our experience with using conditions shows that they can sometimes be quite simple and still be useful.

Network Generation

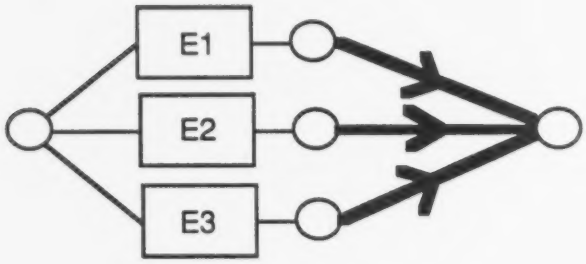
Sequences. Generation of a network for a sequence of patterns proceeds as follows. Suppose we have obtained a subnet for the first expression in the sequence. To ensure that the patterns specified by the consecutive subexpressions are matched in order, we need only build the second subnet so that its start node is the end node for the first subnet. We likewise let each subsequent pair of adjacent component networks share end and start nodes. The start node for the whole network is that of the sequentially first component network and the end node of the whole is that of the last component network. This construction is shown in Figure 1 for the sequential pattern {E1 E2 E3}. Each box represents an arbitrarily complicated expression. All that has to be known about them in order to incorporate these expressions into a more complex net is that they have a single start and stop node as shown. Note how the stop node of E1 is the same as the start node of E2, as suggested by the overlapping circles. The correctness of this construction depends on the fact that the subnets are "one-way" nets, in that control can never flow backwards from the stop node to the start node. If this were not the case, the net might recognize the first subexpression, then the second, then wander back and redo the first.

Alternation. To build a network for an alternation we use a single new start node as the subnet start node for each of the subexpressions. We then add T-test arcs (i.e. test arcs for which the condition always evaluates to TRUE) from the various end-nodes to a single new end-node created for the composite net. This is shown in Figure 1 for the alternation {E1 / E2 / E3}. The lighter arcs represent the nets previously generated for E1, E2, and E3. The arcs added to implement the alternation are shown as heavy arcs. Clearly the newly constructed network can only be traversed if exactly one of the component networks can be satisfied.

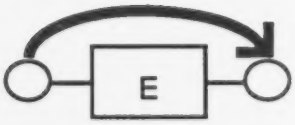
{ E1 E2 E3 }



{ E1 / E2 / E3 }



- E



+ E

* E

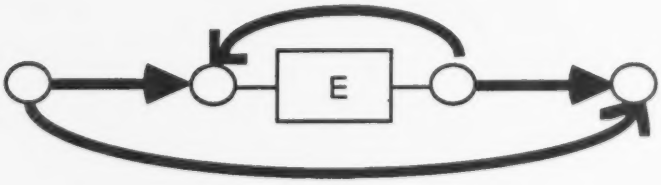


Figure 1 Network Implementation of HGSL constructs. Boxed "E" and thin arrows are the previously constructed net of the arbitrary expression E.

Optional expressions. Given an expression and some net that implements it, we can easily add arcs to make the same expression optional. We do this by adding a T-test arc, evaluated after the first arc of the expression, leading from the start node to the end-node. We then have the option either to pass through the net, or to match nothing to it, as shown in Figure 1.

Repetitions. Implementing one-or-more repetition is a bit complicated. Assuming we have generated a net for the expression to be repeated, we add an arc leading from its end node back to its start node, as shown in Figure 1. This will allow the pattern to be matched more than once. We also create a new end node for the composite network and connect the component network's end node to it via a T-test. After the pattern has been matched one or more times, control can follow this path out of the network.

The most complicated network construction is that for zero-or-more repetition. As shown in Figure 1, we take the network of the expression to be repeated and add a backward T-test from its end to its start node. As in one-or-more repetition, this allows the pattern to be matched more than once. We also create new start and end nodes. The old end node is connected to the new one by a T-test arc. This is the exit from the network after the pattern has been matched one or more times. Finally we add a pair of new arcs out of the new start node. The first leads into the old start node. Any path through the network which begins by taking this arc will have to satisfy the repeated pattern one or more times. The second arc is a T-test leading directly to the end node of the composite network. This allows for zero repetitions of the expression.

It can be proven that the network implementations of the HGSL constructs adopted here are correct. However it is also true that the current HGSL compiler does not produce the most compact networks possible. For example, Figure 2 shows a more efficient network construction for alternation.

IMPLEMENTATION ALGORITHMS

The HGSL Compiler

Here we describe how HGSL constructs are compiled into networks suitable for the interpreter described below. The expression to be compiled as a network is parsed by the set of mutually recursive functions shown in Table 2. Each function is responsible for parsing the structure for which it is named and adding the appropriate arcs and nodes.

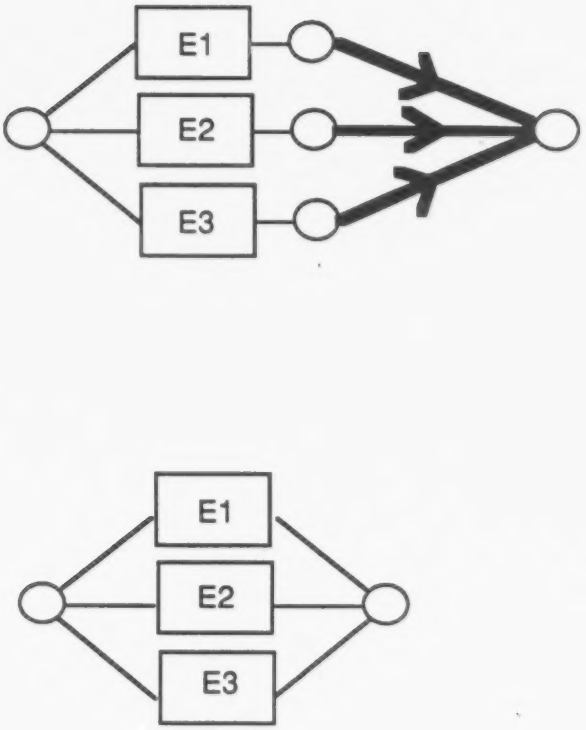


Figure 2. Alternate network implementations of { E1 / E2 / E3 }. The higher one is currently implemented, but the lower one is more efficient.

Table 2

The HGSL Compiler

```
-----------------------------------------------------------------

function #HGSLC(INPUT_FILE: FILE) returns BOOLEAN is
 SELECT OUT PORTION OF FILE TO BE COMPILED;
 return #GRAMMAR;
end #HGSLC;

function #GRAMMAR returns BOOLEAN is
 loop
  if CURRENT-WORD = END-GRAMMAR then
    return TRUE;
  else if #NETWORK-DEFINITION then do nothing;
  else
   return FALSE;
  end if;
 end loop;
end #GRAMMAR;

function #NETWORK-DEFINITION returns BOOLEAN is
 if CURRENT-WORD = NET-DEF then ADVANCE-WORD; end if;
 RECORD CURRENT-WORD AS NAME OF THIS NETWORK;
 ADVANCE-WORD;
 START = A START NODE FOR THIS NETWORK;
 RECORD START AS FIRST NODE OF THIS NETWORK;
 if #EXPRESSION(START,STOP) then
  ADD A POP ARC BEGINNING AT STOP;
  return TRUE;
 else
  return FALSE;
 end if;
end #NETWORK-DEFINITION;

function #EXPRESSION(START,STOP) returns BOOLEAN is
 if CURRENT-WORD STARTS WITH "#" then
   return #NET-CALL(START,STOP); end if;
 if CURRENT-WORD STARTS WITH "<" then
   return #CONDITION(START,STOP); end if;
 if CURRENT-WORD STARTS WITH A LETTER then
   return #LEX-TEST(START,STOP); end if;
 if CURRENT-WORD STARTS WITH "!" then
   return #WORD-TEST(START,STOP); end if;
 if CURRENT-WORD IS "-" then
   return #OPTIONAL(START,STOP); end if;
 if CURRENT-WORD IS "+" then
   return #ONE-OR-MORE(START,STOP); end if;
```

(table continues)

```
 if CURRENT-WORD IS "*" then
   return #ZERO-OR-MORE(START,STOP); end if;
 if CURRENT-WORD IS "{" then
   return #LIST-NO-PREFIX(START,STOP); end if;
  return FALSE;
end #EXPRESSION;

function #NET-CALL(START,STOP) returns BOOLEAN is
 STOP = A NEWLY ALLOCATED NODE;
 USE CURRENT-WORD TO LOOK UP START NODE FOR INVOKED NET;
 ADD NET ARC FROM START TO INVOKED NET WITH NEXT STATE = STOP;
 return TRUE;
end #NET-CALL;

function #CONDITION(START,STOP) returns BOOLEAN is
 STOP = A NEWLY ALLOCATED NODE;
 ADD TEST ARC FROM START TO STOP USING CURRENT-WORD
  AS TEST EXPRESSION;
 return TRUE;
end #CONDITION;

function #WORD-TEST(START,STOP) returns BOOLEAN is
 STOP = A NEWLY ALLOCATED NODE;
 ADD WORD-TEST ARC FROM START TO STOP USING CURRENT-WORD
  AS WORD TO COMPARE WITH;
 return TRUE;
end #WORD-TEST;

function #LEX-TEST(START,STOP) returns BOOLEAN is
 STOP = A NEWLY ALLOCATED NODE;
 ADD LEX-TEST ARC FROM START TO STOP USING CURRENT-WORD
  AS LEXICAL CATEGORY TO COMPARE WITH;
 return TRUE;
end #LEX-TEST;

function #OPTIONAL(START,STOP) returns BOOLEAN is
 WORD-ADVANCE;
 if #LIST-NO-PREFIX(START,STOP) then
  ADD "T" TEST ARC FROM START TO STOP;
  return TRUE;
 else
  return FALSE;
 end if;
end #OPTIONAL;
```

(table continues)

```
function #ONE-OR-MORE(START,STOP) returns BOOLEAN is
 WORD-ADVANCE;
 if #LIST-NO-PREFIX(START,STOP2) then
  STOP = NEWLY ALLOCATED NODE;
  ADD "T" TEST ARC FROM STOP2 TO START;
  ADD "T" TEST ARC FROM STOP2 TO STOP;
  return TRUE;
 else
  return FALSE;
 end if;
end #ONE-OR-MORE;

function #ZERO-OR-MORE(START,STOP) returns BOOLEAN is
 ADVANCE-WORD;
 START2 = NEWLY ALLOCATED NODE;
 if #LIST-NO-PREFIX(START2,STOP2) then
  STOP = NEWLY ALLOCATED NODE;
  ADD "T" TEST ARC FROM START TO START2;
  ADD "T" TEST ARC FROM STOP2 TO STOP;
  ADD "T" TEST ARC FROM START TO STOP;
  ADD "T" TEST ARC FROM STOP2 TO START;
  return TRUE;
 else
  return FALSE;
 end if;
end #ZERO-OR-MORE;

function #LIST-NO-PREFIX(START,STOP) returns BOOLEAN is
 ADVANCE-WORD;
 STOP =  A NEWLY ALLOCATED NODE;
 if #EXPRESSION(START,STOP2) then null; else return FALSE; end if;
 if CURRENT-WORD = "/" then
  ADD "T" TEST ARC FROM STOP2 TO STOP;
  until CURRENT-WORD = "}" loop
   if #EXPRESSION(START,STOP2) then
    ADD "T" TEST ARC FROM STOP2 TO STOP;
   else return FALSE;
   end if;
  end loop;
  return TRUE;
 else
  until CURRENT-WORD = "}" loop
   START = STOP2;
   if #EXPRESSION(START,STOP2) then null;
   else return FALSE; end if;
  end loop;
  STOP = STOP2;
  return TRUE;
 end if;
end #LIST-NO-PREFIX;
```

#HGSLC is the top-level function of the compiler. It takes the name of an input file and asks the user whether the entire grammar should be compiled or if just one of the network definitions should be recompiled. It then calls #GRAMMAR. Since a grammar is just a list of network definitions ended by the key word END-GRAMMAR, the function #GRAMMAR calls #NETWORK-DEFINITION repeatedly until that key word is encountered.

#NETWORK-DEFINITION checks for the key word NET-DEF, records the name of the network being defined, and then calls #EXPRESSION. #EXPRESSION is simply a large select statement which examines the current character to determine which type of expression follows. The appropriate function is then called. The functions #NET-CALL, #CONDITION, #LEX-TEST, #WORD-TEST are low-level functions which actually build single arcs for the routines that call them.

#OPTIONAL, #ONE-OR-MORE, #ZERO-OR-MORE, and #LIST-NO-PREFIX are intermediate level routines that all call the function #EXPRESSION one or more times, adding additional arcs to the results of these function calls, and sometimes piecing them together to form a larger net. #OPTIONAL adds a single arc to whatever structure has been built for its component expression. #ONE-OR-MORE adds two arcs and #ZERO-OR-MORE adds four. #LIST-NO-PREFIX is the most complicated net, since it builds either a disjunctive or a sequential net and in either case this requires piecing together the nets generated during calls to #EXPRESSION as discussed above and pictured in Figure 1.

The ATN Interpreter

The interpreter used by our system is fairly conventional. The output of the compiler is a set of networks based on the constructions given above. These networks are represented as sets of arcs leading from one node to another. The interpreter has only one major data structure, a stack of nodes that is used to maintain the current path through the various nets. The interpreter repeatedly pops this stack and tries to extend the path, generally by evaluating the next arc out of the most recently stacked node.

The output produced by the HGSL system is a syntax tree such as that shown in Figure 3. This tree is based on the parse path constructed automatically by the interpreter. Once parsing the top-level net has been successfully completed, the parse path will be stored at the top of the parse stack and can be interpreted as a syntax tree.

Table 3 gives the interpreter algorithm in pseudo-code. The stack frame, declared in lines 2-8, contains a node id number, the id number of the last outgoing arc examined, the position of

```
(#START
    (#SENTENCE
        (#NP
            (DET  THE)
            (NOUN INSTRUCTOR))
        (#VP
            (VERB  PERFORMED)
            (#NP
                (DET THE)
                (NOUN PROCEDURE)))))
```

Figure 3. Example of parser output for the sentence "The instructor performed the procedure."

Table 3

The ATN Interpreter

```
 1  function ATN_INTERPRETER returns SUCCESS-OR-FAILURE is
 2   FRAME is record
 3    STATE: integer;
 4    LAST-ARC-TESTED: integer;
 5    POSITION-IN-SENTENCE: integer;
 6    ACTIVE-NET-CALL-FLAG: boolean;
 7    NET-PATH: list of FRAME;
 8   end record;
 9   CURRENT-FRAME: FRAME;
10   S: stack of FRAME;
11   POPPED-SENTENCE: boolean;
12  begin
13   POPPED-SENTENCE := FALSE;
14   PUSH(S,[#INVALID-NET-NAME,1,0,TRUE,NIL]);
15   PUSH(S,[#START1,0,1,FALSE,NIL]);
16   while not POPPED-SENTENCE loop
17    CURRENT-FRAME = POP(S);
18    if CURRENT-FRAME MATCHES [-,-,-,TRUE,NET-PATH] then
19     S := PUSH(S,NET-PATH);
20    else if CURRENT-FRAME MATCHES [-,-,-,TRUE,NIL] then
21     S := PUSH(S,[-,-,-,FALSE,NIL]);
22    else if CURRENT-FRAME MATCHES
23     [STATE,ARC-NO,POSITION,FALSE,NIL] then
24     NEXT-ARC-NO := ARC-NO + 1;
25     NEXT-ARC := FETCH-ARC(STATE,NEXT-ARC-NO);
26     TEST-RESULT := TEST-ARC(NEXT-ARC,POSITION);
27     if TEST-RESULT = RAN-OUT-OF-ARCS then
28      if STATE = INVALID-NET-NAME then
29       return FAILURE;
30      end if;
31     else if TEST-RESULT = SUCCESS then
32      if ARC-TEST MATCHES (ARC-TYPE,X,NEXT-STATE) and
33       MEMBER-OF(ARC-TYPE,{WORD,LEX}) then
34       S = PUSH(S,[NEXT-STATE,0,POSITION+1,FALSE,NIL]);
35      else if ARC-TEST MATCHES (TEST,X,NEXT-STATE) then
36       S = PUSH([NEXT-STATE,0,POSITION],S);
37      else if ARC-TEST MATCHES (POP,NIL,NIL) then
38       INVOKER-FRAME := MOST RECENTLY STACKED FRAME
39        THAT MATCHES [STATE,ARC,-,TRUE,NIL];
40       NET-PATH := ALL FRAMES ABOVE INVOKER-FRAME;
41       if STATE = #START1 then POPPED-SENTENCE := TRUE; end if;
42       ARC-TEST := FETCH-ARC(STATE,ARC);
43       MATCH ARC-TEST TO [ARC-TYPE,NIL,NEW-STATE];
44       S := PUSH(S,[STATE,ARC,X,FALSE,NET-PATH]));
```

(table continues)

```
45        S := PUSH(S,[NEW-STATE,0,X,FALSE,NIL]);
46       end if;
47      else
48       S := PUSH(S,CURRENT-FRAME);
49      end if;
50     end if;
51    end loop;
52    return SUCCESS;
53   end ATN_INTERPRETER;
```

---

the parser in the current sentence when the node was reached, a flag signalling whether the last arc evaluated triggered a currently active net call, and the network path of any completed net call originating at the node. Initially the stack contains two frames, the node #INVALID-NET-NAME which is shown invoking the first node of the top-level network #START (lines 14,15). The standard cycle of the interpreter is a loop (lines 16,50) which exits either with success when the top-level net call to #START returns, or with failure when the stack is exhausted.

The loop begins with popping the stack (line 17). The popped stack frame will fall into one of four categories. The first case (line 18) is when the stack frame has a completed net path stored with it. In this case, the path is removed from the current frame and placed on the stack allowing the interpreter to back up into a previously completed net-call. The second case (line 20) is for stack frames which have triggered a currently active net call. Coming across one of these indicates that the net call has failed, so after the ACTIVE-NET-CALL-FLAG is set to FALSE, the stack frame is pushed to allow the next arc for that node to be considered.

The third case (line 22) is for all other frames and involves fetching and testing the next outgoing arc (lines 24-25). If there are no more arcs, then the current node will be given no further consideration. If there are no more arcs and the current node is the one that marks the stack bottom, (i.e. #INVALID-NET-NAME), then the interpreter returns FAILURE (lines 28-29). Assuming there is an arc to evaluate, and that evaluation succeeds (line 31), the appropriate action is taken depending on the arc. If evaluation fails (line 47), no action is taken before beginning the next cycle when the next arc will be examined.

A successful word or lex test will require stacking a frame for the next state together with a current word position that has been incremented by one (lines 32-34). A successful conditional test also causes the next state to be stacked (line 35-36). A pop arc is always successful and its processing involves looking

down in the stack for the most recent active net call (line 38-39). All nodes stacked above this represent the path which has been found through the invoked net and are added to the stack frame of the node which triggered the net call.

If an invocation of the top-level network was popped (line 41), then POPPED-SENTENCE will be set to TRUE, control will exit the loop, and the interpreter will return SUCCESS. Note that because of this, the HGSL supplied by the user must not be recursive at the top level, i.e. the grammar must not include a call to #START.

A GRAMMAR FOR TECHNICAL PROSE

A substantial grammar has been developed using HGSL. This section presents the grammar and discusses its coverage and some of its strong and weak points.

The grammar was developed to cover a set of sentences taken from Navy technical training materials. Any sentence which could not be parsed lead to an extension of the HGSL grammar. HGSL was designed to make this process as quick as possible by allowing nets to be described in a compact, easily-read formalism. In practice the system did make improvements easier. HGSL also makes it easier to spot rules which are either inconsistent or not as general as they might be. Far-ranging reorganizations of the grammar, while very time-consuming for an explicit node and arc representation, are fairly simple with a powerful grammar shorthand like HGSL.

GRAMMAR DESCRIPTION

Table 4 presents an overview of the grammar; Table 5 lists the lexical categories used by the grammar. The complete HGSL text appears as an appendix; in this section the text will be presented for description piece-meal. One thing to note about this grammar is that it was developed to recognize rather than to generate sentences. Consequently it would not be difficult to use it to generate some very bad sentences. Also since it is meant to be suitable for systems which critique poor prose (see Kieras, 1985), the nets should not fail on sentences which are only slightly ungrammatical; otherwise the majority of the input text might never survive the first-stage syntactic analysis.

The grammar developed here suffers from an ad hoc approach to conjunction which has lead to the inclusion of conjunctive branches in many of the nets. The option which we did not pursue is to build a special mechanism outside the grammar that would have constituted a general theory of where conjunction can occur,

Table 4

Network Names and General Characterizations

#START
The top-level network and starting point for all parsing.

#HEADING
Titles, chapter or paragraph headings, etc consisting of some formatting mark (i.e. indentation, roman numerals) followed by a noun phrase.

#CSTATEMENT
Conjunctive statement. One or more sentences conjoined together.

#STATEMENT
Either declarative or imperative.

#DECLARATIVE-STATEMENT

#IMPERATIVE-STATEMENT

#PPCL
Past participle clause. Clauses based on a past participle and explicitly introduced by some subordinating conjunction such as when given aircraft type and weather conditions.

#VINGCL
Present participle clause. Clauses based on a present participle introduced by a subordinating conjunction. While collecting and safeguarding drug evidence.

#SUBCL
Subordinate clause. Full sentences introduced by a subordinating conjunction. Because the procedure was performed incorrectly, ...

#VERB-COMPLEX-ARGUMENT
Infinitive clauses following "to" which act like the object of a verb or simply give the purpose of the action. He tried to perform the procedure He performed the procedure to conform with regulations

(table continues)

#FOR-TO
Infinitive clauses which begin a sentence To perform the procedure , ... For the students to perform the procedure...

#WHETHER-OR
Compound condition formed with "whether". Whether the user is a novice or if he knows the system well, this reference manual will be helpful.

#SUBRELCLS
Subject relative clause. The modified noun is the subject of the relative clause.

#ELIDED-VP
Elided verb phrase. Clauses based on a past or present participle from which a form of the verb "to be" has been deleted. May introduce a sentence or modify a noun. Given adequate instructions, the students ... a procedure requiring expert supervision

#ADJCL
Adjective clause. Clauses which follow and complete the meaning of certain adjectives. able to perform the procedure

#OBJRELCLS
Object relative clause. The modified noun is the object of the relative clause which follows. the procedure the instructor performed

#FOR-TO-RELCLAUSE
A relative clause based on an infinitive and possibly introduced by "for". the equipment for the trainees to use during class

#COMPOUND-MODIFIER
Conjunction of noun-modifying phrases sometimes following the modified noun, but sometimes preceding it. "Whether working with a visual information specialist or alone, ... All procedures, official or unofficial, ...

#CVP
Conjoined verb phrase.

#VP
Verb phrase.

(table continues)

#VCOMP
Verb complement. The modifying phrases that follow the main verb and other verb parts such as participles and the infinitive.

#GINF
Generalized infinitive. Includes not only the lexical category INF, but multiple word infinitives followed by modifying phrases such as The procedure is to be performed by the instructor.

#CNP
Conjoined noun phrases.

#NP
Noun phrase. Both the usual noun phrase consisting of adjectives and head noun as well as whole clauses which can function like a noun, e.g. What the instructor said was unclear.

#GERUND
A present participle and its modifier acting as a noun. Following the instructions for this procedure is crucial.

#PREPPHR
Prepositional phrase. Also allows for conjunctions as in With the instructor's assistance and in keeping with the rules ...

#RELCLAUSE
Relative clauses. Either subject (SUBRELCLS) or object (OBJRELCLS) relative clauses.

#INTERRUPTER
Phrases typically set off by commas and serving to qualify a noun phrase.

Table 5

Table of Lexical Categories

ADJCL	Adjective introducing a clause - Available for ...
ADJ	Adjective
ADV	Adverb
AUX-DO	Forms of the verb do acting as verb auxiliaries
AUX-HAVE	Forms of the verb have used for the past tense
AUX-IS	Forms of the verb is in progressives or passives
AUX-MODAL	Modals such as may, might, should, etc.
CONJ	Conjunction
DEFDET	Definite determiner
HEADING-MARK	Characters marking a title or heading
INF	Infinitive form of a verb
NAME	Proper name
NDEFDET	Indefinite determiner
NEG	Negative
NOUN	Noun
POSS-MARK	Apostrophe in possessive forms
PPCL	Words introducing a past participle clause - though inspected by the instructor
PREDETADJ	Adjective preceding a determiner - all the best
PREP	Preposition
PRN	Pronoun
PROPPRN	Propositional pronoun - That is not true.
RELPRN	Relative pronoun
RESRELPRNR	Restrictive relative pronoun - that as opposed to which
SUBCL	Words that introduce a subordinate clause
VERB	Any form of a verb, inflected or not
VERBING	Present participle
VERBPP	Past participle
VINGCL	Words that introduce a present participle clause - while performing the procedure

and thus would have saved us the effort of addressing the problem on a case-by-case basis. Such an approach has the disadvantage of making the ATN interpreter non-standard.

The top-level network is #START, shown in Table 6. Since the grammar is meant to parse technical prose, rather than isolated sentences, #START recognizes both sentences (#CSTATEMENT) and any headings which may occur in the passage. Headings are assumed to be indicated by format markings such as peculiar indentation or special text editor control characters and consist of a noun phrase in the broadest sense (#CNP). #CNP includes any phrase that could be the subject of a sentence. This allows the system to handle titles such as "How to perform the procedure". For convenience of discussion, the rest of the grammar is divided into the following groups: sentences, subordinate clauses, relative clauses, verb phrases, and noun phrases.

Sentences

The main sentence patterns (see Table 6) are #DECLARATIVE-STATEMENT and #IMPERATIVE-STATEMENT. The first is simply a noun phrase followed by a verb phrase, while the second consists only of a verb phrase. The various clauses which can introduce or follow these basic sentence patterns are common to the two, and are factored off into the higher-level net #STATEMENT. Since #STATEMENT is the first definition of any complexity that we have so far encountered, it may be helpful to interpret it in some detail.

A #STATEMENT begins with zero or more instances of the six types of introductory clause, optionally followed by a comma, then proceeding to either a declarative or an imperative sentence, in either case optionally followed by a subordinate clause. #STATEMENT is in turn the main constituent of #CSTATEMENT which allows conjunction of two or more simple sentences. This net is fairly subtle. It begins with a #STATEMENT. Since the next expression is preceded with a "-", we know it may also end with that first #STATEMENT. Alternately we can add one or more comma-#STATEMENT pairs, before closing with a CONJ (possibly preceded with a comma) and one last #STATEMENT. There are two other nets which are entirely devoted to describing the conjunction of simpler expressions, one for conjoined nouns (#CNP) and one for conjoined verbs (#CVP).

Table 6

HGSL for #START and Sentences

```
NET-DEF #START
    { #HEADING /
      { #CSTATEMENT !. } }

NET-DEF #HEADING
    { HEADING-MARK #CNP }


NET-DEF #CSTATEMENT
    { #STATEMENT
      - { * { !, #STATEMENT }
              - { !, }
              CONJ
              #STATEMENT
          }
    }

NET-DEF #STATEMENT
    { * { #FOR-TO /
          #WHETHER-OR /
          #ELIDED-VP /
          #PREPPHR /
          #SUBCL /
          #ADV }
      - { !, }
      { #DECLARATIVE-STATEMENT / #IMPERATIVE-STATEMENT }
      - { SUBCL } }

NET-DEF #DECLARATIVE-STATEMENT
   { #CNP #CVP }

NET-DEF #IMPERATIVE-STATEMENT
   #CVP
```

Subordinate Clauses

There is considerable variety among the subordinate clauses. #PPCL and #VINGCL, shown in Table 7, are nets based on past and present participles, respectively. Each is introduced by a subordinating conjunction. Note how the incremental approach to grammar design has lead to an option for conjunction in the #VINGLCL, but not in #PPCL. This is because the sample sentences so far processed have not required conjunction in #PPCL. #SUBCL is the combination of a subordinating conjunction and a full sentence.

#VERB-COMPLEX-ARGUMENT is an infinitive-based phrase appearing after the verb. Sometimes it will be identifiable as an argument of the verb, thus justifying its name, as in He hopes to get back to work. In other cases it will be modify the meaning of the entire sentence by giving the reason for which some action was taken, as in He did it to better his chances.

#FOR-TO is used to introduce a sentence. Like #VERB-COMPLEX-ARGUMENT, it is based on the infinitive, but allows the subject of the infinitive to be specified by adding for and a noun phrase at the front (For the plan to work, ...). #WHETHER-OR is probably best thought of as a complex subordinate clause, since it combines two #STATEMENTs into a subordinate relation to the main sentence, as in Whether the result is positive or if it cannot be determined, ... On the other hand, #WHETHER-OR differs from #SUBCL in that it may be based on a sentence fragment rather than a complete sentence as in Whether old or new, ... By comparison, Because old or new ... is unacceptable.

Relative Clauses

The relative clauses shown in Table 8 can be used to modify nouns, or in some cases, can be used in the place of nouns. #SUBRELCLS includes relative clauses in which the noun modified plays the role of subject in the relative clause. The most obvious variety uses a relative pronoun as in The procedure that works. One type of #SUBRELCLS that does not use relative pronouns is the #ELIDED-VP, which is based on a past or present participle, e.g. the procedures studied in this course or the trainees having the most difficulty, for the above examples. They are called elided verb phrases because they are taken to be shortened forms, such as the procedures which were studied in this course and the trainees who are having the most difficulty. Like most clauses which can modify a noun, #ELIDED-VP can be shifted to the front of the sentence, in which case it is being used to describe the subject of the sentence, as in Elected for the first time in 1982, the congressman ... .

Table 7

HGSL for Subordinate Clauses

```
NET-DEF #PPCL
  { PPCL VERBPP #VCOMP }

NET-DEF #VINGCL
  { VINGCL #GERUND
         *  { { CONJ / !, } #GERUND } }

NET-DEF #SUBCL
  { SUBCL #STATEMENT }

NET-DEF #VERB-COMPLEX-ARGUMENT
  { !TO #GINF }

NET-DEF #FOR-TO
  { - { { !FOR #CNP } / { !IN !ORDER } }
    !TO #GINF }

NET-DEF #WHETHER-OR
  { !WHETHER
    { #COMPOUND-MODIFIER /
       { #STATEMENT - { !OR !IF #STATEMENT } } }

NET-DEF #COMPOUND-MODIFIER
  { - { !BOTH / !EITHER }
     { ADJ / #ELIDED-VP }
     * { { CONJ / !, } { ADJ / #ELIDED-VP } } }
```

Table 8

HGSL for Relative Clauses

```
-------------------------------------------------------------------

NET-DEF #SUBRELCLS
  { { RESRELPRN #CVP } /
      #PREPPHR /
      #ELIDED-VP /
      #ADJCL }

NET-DEF #ELIDED-VP
  { - { NEG } * { ADV }
     { #GERUND / { VERBPP #VCOMP } } }

NET-DEF #ADJCL
  { - { !, }
   ADJCL { #PREPPHR / { !TO #GINF } } }

NET-DEF #OBJRELCLS
  { - { RESRELPRN } #DECLARATIVE-STATEMENT }

NET-DEF #FOR-TO-RELCLAUSE
  { - { !FOR #CNP } !TO #GINF }

-------------------------------------------------------------------
```

Yet another variety of #SUBRELCLS is the #ADJCL. It is based on an infinitive or prepositional phrase and introduced by certain adjectives such as eager, impatient, or glad as in Trainees glad to complete their instruction or Available to all employees, group insurance ... .

Aside from #SUBRELCLS, the other major varieties of relative clause are #OBJRELCLS and #FOR-TO-RELCLAUSE. In #OBJRELCLS the modified noun plays the role of the object and the relative pronoun is optional, as in The procedure [that] the instructor demonstrated. #FOR-TO-RELCLAUSE is a restricted version of #FOR-TO appropriate for use as a relative clause. It allows us to handle the thing to do or the thing for you to do.

Verb Phrases

There are three major verb phrase nets shown in Table 9, the principal one being #VP, which generates verb forms. It includes a fairly careful description of verb formats, covering the use of modal auxiliaries such as can or may, the use of do, be and have as auxiliaries, and simple tensed verbs, with consideration given to negation and adverbs occurring between

Table 9

HGSL for Verb Phrases

```
NET-DEF #CVP
    { #VP
      * { + { CONJ / !, }
            #VP
        }
    }

NET-DEF #VP
    { * { ADV }
      { { AUX-MODAL - { NEG } * { ADV } #GINF #VCOMP } /
        { AUX-DO - { NEG } * { ADV } INF #VCOMP } /
        { - { NEG } AUX-IS * { ADV }
              { VERBING / VERBPP } #VCOMP
              * { CONJ { VERBING / VERBPP } #VCOMP } } /
        { AUX-HAVE - { NEG } * { ADV } VERBPP #VCOMP } /
        { VERB #VCOMP } } }

NET-DEF #VCOMP
  *  { #CNP /
        { < LAST WORD IS A VERB
           TAKING STATEMENT OBJECT > #CSTATEMENT } /
        { !THAT #CSTATEMENT } /
        { { - { !, } #PREPPHR
           * { CONJ #PREPPHR } } /
        ADV /
        ADJ /
#PPCL /
#VINGCL /
#VERB-COMPLEX-ARGUMENT /
{ !, #INTERRUPTER !,  } /
{ !{ #INTERRUPTER !}  }  }  }

NET-DEF #GINF
  {  *  {   ADV  }
     {  {  INF  #VCOMP  }  /
        {  !HAVE  VERBPP  #VCOMP  }  /
        {  !HAVE  !BEEN  {  VERBPP / VERBING }  #VCOMP  }  /
        {  !BE  * { ADV }  { VERBPP / VERBING }  #VCOMP
           * { CONJ  { VERBPP / VERBING }  #VCOMP }  }  }  }
```

parts of the verb. Since this is a complicated definition, we will give several examples of the verb phrases that it includes.

There are basically five alternatives based on AUX-MODAL, AUX-DO, AUX-IS, AUX-HAVE, or VERB. The AUX-MODAL case begins with a modal verb such as may, might, or should, and then continues on to some infinitive and whatever object may follow the infinitive (#VCOMP). As an example, consider must perform the procedure. Adverbs and negating elements can be interspersed as indicated to give must not carelessly perform the procedure. The AUX-DO alternative is quite similar, but is based on a form of the verb to do, rather than a modal. The sort of infinitive phrase which follows to do is also slightly more restrictive, INF as opposed to #GINF. #GINF can generate have done, whereas the lexical category INF cannot. As a consequence, should have done it is allowed, but did have done it is not.

AUX-IS verb phrases have some form of the verb to be followed by a participle, either present (VERBING) or past (VERBPP). The participle can then be followed by the usual objects. Examples of this would be been performing the procedure or was told by the instructor. As indicated by the latter example, AUX-IS verb phrases include some passive voice constructs. The AUX-IS verb phrase has been extended to allow building conjunctive verb phrases by adding a CONJ and a second participle to give phrases such as been cleaned and inspected.

AUX-HAVE generates the past tense with have and so requires a past participle. An example with the optional arguments is have performed the procedure. The VERB-based verb phrase is the simplest pattern and captures the present tense of simple verbs, e.g. perform the procedure.

Every basic verb phrase described in #VP ends with a #VCOMP, a net describing the numerous phrases which can follow the main verb of a sentence. The first alternative in #VCOMP is the direct object (#CNP). The second alternative uses a condition (note the angle brackets). The rational for this condition is as follows. Some verbs take whole clauses as their objects, as in I hope they all get here on time. It would be very inefficient to begin parsing a sentence after every verb, so the condition ensures that we attempt this only if the verb is one of the relatively few which can take clauses for their objects. In the third #VCOMP option, a clause once again serves as object of the sentence, but it is explicitly marked by that, as in I hope that they all get here on time.

Some of the more straightforward post-verb elements include one (or more) prepositional phrases and the lexical class ADV (adverbs). The lexical class ADJ (adjectives) is appropriate only after such verbs as be, seem, become, etc. Currently no

attempt is made to implement this restriction, which could be done with a condition.

#PPCL and #VINGCL are also possible verb complements, as in He executed the procedure as ordered by the instructor or He executed the procedure before realizing it was inapplicable. As mentioned above, #VERB-COMPLEX-ARGUMENT accounts for infinitive phrases that follow the verb.

The last options given under #VCOMP are for interrupter phrases, i.e. those which are likely to be set off from the rest of the sentence by commas or brackets. Since our understanding of the relation of these phrases to the rest of the sentence is incomplete, this constituent has an undeniable catch-all flavor, generating noun phrases, conjoined adjectives, and at least some subordinate clauses. Giving a more satisfactory account of these phrases would be an important next step for this grammar.

Likewise, the #VCOMP net could be improved by taking into account the few sequential restrictions that govern the ordering of the structures included. For example, a verb complement may include both a noun phrase and a clause object, but the order is not arbitrary, as shown by the contrast between The officer told the trainee his promotion was approved and *The officer told his promotion was approved the trainee. Currently #VCOMP does not impose any such restriction. A #VCOMP consists simply of zero or more items from the list of possible phrase structures. The surprising thing about this net is that it works as well as it does.

The last major verb net is #GINF which covers infinitive forms and likewise makes use of #VCOMP to describe complete infinitive-based phrases. There are basically four phrases here which can be illustrated by the following examples: do something, have done something, have been doing something, and be doing something.

Noun Phrases

The major noun phrase net is #NP, shown in Table 10. The first two options of the net are the most complicated. The first describes the sort of clause which can function both as a relative clause and as a noun, as in What he saw amazed him. There is also an interesting conjunctive version, Do you know where or when this trend started? As is clear from these examples, RELPRN is a fairly broad class including when, where, how, which, that, and so forth. It is not essential to give a full sentence after the RELPRN to get one of these noun-replacing clauses. An infinitive phrase will do just as well: Does he know where to turn?

Table 10

HGSL for Noun Phrases

```
NET-DEF #CNP
  { - { !BOTH / !EITHER / !NEITHER }
      { #NP
        * { + { !, / !; / CONJ } #NP }
        - { !, } } }

NET-DEF #NP
  { { RELPRN * { + { CONJ / !, } RELPRN }
          #DECLARATIVE-STATEMENT } /
      { RELPRN * { + { CONJ / !, } RELPRN }
          !TO #GINF #VCOMP } /
      { !WHETHER #DECLARATIVE-STATEMENT
        !OR  { !NOT / #DECLARATIVE-STATEMENT } }
      { * { PREDETADJ }
        - { DEFDET / NDEFDET }
        * { NOUN / { ADJ - { CONJ } } }
        NOUN
        - { #RELCLAUSE /
              { POSS-MARK #CNP } /
              { !{ #CNP !} } } } /
        NAME /
        PROPPRN /
        PRN /
        #GERUND }

NET-DEF #GERUND
  { - { NEG } { VERBING / !HAVING VERBPP } #VCOMP }

NET-DEF #PREPPHR
  { PREP #CNP
      * { + { CONJ / !, } PREP #CNP } }

NET-DEF #RELCLAUSE
  { #FOR-TO-RELCLAUSE / #SUBRELCLS / #OBJRELCLS }

NET-DEF #INTERRUPTER
  { #COMPOUND-MODIFIER /
    #CNP /
    #PPCL }
```

The #WHETHER-OR clause discussed above in its role as a sentence-introducing dependent clause can also serve in place of a noun. For example, Whether the procedure is efficient is not crucial. The most common pattern described by #NP is the more obvious grouping of nouns and adjectives ending with a head noun, and then possibly followed by relative clauses. This pattern also covers noun phrases that turn out to be possessive forms. Other possible noun forms include proper names, pronouns (both PROPPRN and PRN) and gerunds (i.e. those phrases based on a present participle but serving as a noun).

GRAMMAR COVERAGE

The grammar was originally developed to handle technical training materials written by Navy writers. The goal was to be able to process early drafts of such material, and not finished versions of the material, because the parser was intended to be used as part of a computerized comprehensible writing aid (see Kieras, 1985). A sample of target materials was collected and supplied by the Naval Personnel Research and Development Center (NPRDC), along with a lexicon containing about 10,000 words, tagged with their traditional parts of speech. This lexicon includes most of the words appearing in military technical training materials. It should be noted that a large quantity of such material appears in essentially an outline format, with heavy use of "telegraphic" prose. We did not attempt to ensure that the grammar could handle such material, both because key parts of the content are conveyed by the outline structure rather than sentence content, and because the telegraphic style is probably inappropriate for such documents anyway. As an indication of the coverage of the grammar, it parses all of the examples shown in Table 11.

Convergence of Coverage

The grammar was originally developed to handle the target materials in the usual non-systematic manner. That is, a few sentences were chosen and tried on the grammar. If there was a failure to parse the sentence, a decision was made whether extending the grammar would be reasonable, and if so, the extension was made. However, we had the usual experience of parser developers in that a lot of syntactic coverage comes very quickly in the development of the grammar, but each extension accounts for fewer new syntactic forms. Thus, when coverage is assessed in terms of the variety of syntactic forms, further work on the grammar tends to produce less and less additional coverage. But if the goal is to handle real material, with realistic distributions of syntactic forms, is it possible that the grammar development process converges to an adequate coverage? Of course, there are too many possible syntactic forms

Table 11

Example sentences from each NPRDC materials sample

Sample 1

Given the logarithm table, a chain of amplifiers and/or attenuators with the gain or loss of each expressed in db, and the input power in watts, compute the gain or loss and output power.

Sample 2

In order to ensure that all art work requests leaving and returning to the IPDD are accurate and the requested word is done to the satisfaction of the customer, the following procedures will be adhered to in submitting audio-visual production requests.

Sample 3

Due to the technical nature of these performance tests and the requirement for the proctor to be fully aware of the examinees' actions and their consequences at all times, it is required that the proctor be qualified to teach this course of instruction.

Sample 4

Identify the proper methods of approaching a drug offender while collecting and safeguarding drug evidence as specified in applicable publications.

---

to hope realistically for a complete grammar. But the question is whether the process would get to a point of diminishing returns at a reasonably high proportion of sentences in the target material that are covered.

Thus, as part of the final grammar development process, a convergence study was conducted. A series of material samples were used, with the grammar extended to handle each sample in turn. A record was kept of each change made in the grammar, so that we could roughly quantify whether the extensions to the grammar either increased or decreased as we went from one sample to the next.

The specific samples were supplied by NPRDC. These were actual samples of draft materials to be used in technical

training. The sentences in these samples had been classified into two groups, based on whether or not they could be simply parsed by an extension of the relatively simple ATN grammar for technical prose found in Kieras (1983). We assumed that all of the sentences that could be parsed by the NPRDC grammar could also be parsed by the current ATN, which like the NPRDC grammar, evolved from the same original simple ATN. We then focused on the sentences that could not be parsed by the NPRDC grammar. The sentences shown in Table 11 are examples of sentences that could not be parsed by the NPRDC grammar, but could be parsed by the current grammar, after it was fully developed to handle these samples. These examples are chosen to represent the more complex sentences that could not be handled by the NPRDC parser, rather than the simpler ones. The samples were used in order of increasing size of the sample, which was the same as the order of increasing number of sentences that could not be parsed by the NPRDC ATN. The grammar was elaborated as required for each of the sentences, and a record kept of how many such changes were made. Notice that the criterion for a successful parse was only that the parser succeeded in producing a parse tree that was not grossly wrong. Such parse trees may have difficulties in terms of semantic interpretation, but we did not make a systematic effort to either quantify the number of such problems or to resolve them.

The results are shown in Table 12. As shown in the Table, the first sample consisted of a total of 23 sentences, 5 of which could not be parsed by the NPRDC ATN, and all 5 of these sentences required extensions to our grammar. The next sample had 30 such non-parsable sentences, and 12 required extensions to the grammar. The fourth sample, however, had a total of 109 sentences in it, 62 of which could not be parsed by the NPRDC grammar, but by the time we reached this fourth sample, only four sentences required extensions to the grammar.

This overall decrease in the number of grammar extensions suggests that the grammar is converging to a coverage of the target materials that would be fairly adequate. Notice that each sample came from a different writer, so that we exposed the parser to the idiosyncracies of different writer's styles. Although this convergence study is very limited, we are encouraged that practically useful parsers for this target material can be developed, and that the grammar presented here is close to being a practically useful parser.

Table 12

Grammar Convergence Results

	Sample in order			
	1	2	3	4
Total sentences in sample	23	46	65	109
Sentences not simply parsed	5	30	39	62
Sentences requiring grammar extensions	5	12	6	4
Percentage of total	22%	26%	9%	4%

References

Kieras, D. E. (1983). A simulation model for the comprehension of technical prose. In G. H. Bower (Ed.), The Psychology of Learning and Motivation, 17. New York, NY: Academic Press.

Kieras, D. E. (1985). The potential for advanced computerized aids for comprehensible writing of technical documents. (Technical Report No. 17, TR-85/ONR-17). University of Michigan

Winograd, T. (1983). Language as a cognitive process: Vol. 1: Syntax. Reading, Massachusetts: Addison-Wesley.

Appendix

The Grammar for Technical Training Materials

```
-----------------------------------------------------------------

NET-DEF #START
    { #HEADING /
   { #CSTATEMENT !. } }

NET-DEF #HEADING
    { HEADING-MARK #CNP }


SENTENCES


NET-DEF #CSTATEMENT
    { #STATEMENT
    - { * { !, #STATEMENT }
        - { !, }
        CONJ
        #STATEMENT
       }
    }

NET-DEF #STATEMENT
    { * { #FOR-TO /
      #WHETHER-OR /
      #ELIDED-VP /
      #PREPPHR /
      #SUBCL /
      #ADV }
   - { !, }
   { #DECLARATIVE-STATEMENT / #IMPERATIVE-STATEMENT }
   - { SUBCL } }

NET-DEF #DECLARATIVE-STATEMENT
   { #CNP #CVP }

NET-DEF #IMPERATIVE-STATEMENT
   #CVP
```

SUBORDINATE CLAUSES

```
NET-DEF #PPCL
  { PPCL VERBPP #VCOMP }

NET-DEF #VINGCL
  { VINGCL #GERUND
       *  { { CONJ / !, } #GERUND } }

NET-DEF #SUBCL
  { SUBCL #STATEMENT }

NET-DEF #VERB-COMPLEX-ARGUMENT
  { !TO #GINF }

NET-DEF #FOR-TO
 { - { { !FOR #CNP } / { !IN !ORDER } }
    !TO #GINF }

NET-DEF #WHETHER-OR
 { !WHETHER
    { #COMPOUND-MODIFIER /
     { #STATEMENT - { !OR !IF #STATEMENT } } }

NET-DEF #COMPOUND-MODIFIER
 { - { !BOTH / !EITHER }
     { ADJ / #ELIDED-VP }
     * { { CONJ / !, } { ADJ / #ELIDED-VP } } }
```

RELATIVE CLAUSES

```
NET-DEF #SUBRELCLS
  { { RESRELPRN #CVP } /
    #PREPPHR /
    #ELIDED-VP /
    #ADJCL }

NET-DEF #ELIDED-VP
  { - { NEG } * { ADV }
   { #GERUND / { VERBPP #VCOMP } } }

NET-DEF #ADJCL
 { - { !, }
   ADJCL { #PREPPHR / { !TO #GINF } } }

NET-DEF #OBJRELCLS
  { - { RESRELPRN } #DECLARATIVE-STATEMENT }
```

```
NET-DEF #FOR-TO-RELCLAUSE
  { - { !FOR #CNP } !TO #GINF }
```

VERB PHRASES

```
NET-DEF #CVP
    { #VP
    * { + { CONJ / !, }
       #VP
       }
    }

NET-DEF #VP
    { * { ADV }
    { { AUX-MODAL - { NEG } * { ADV } #GINF #VCOMP } /
       { AUX-DO - { NEG } * { ADV } INF #VCOMP } /
       { - { NEG } AUX-IS * { ADV }
          { VERBING / VERBPP } #VCOMP
          * { CONJ { VERBING / VERBPP } #VCOMP } } /
       { AUX-HAVE - { NEG } * { ADV } VERBPP #VCOMP } /
       { VERB #VCOMP } } }

NET-DEF #VCOMP
  *  { #CNP /
       { < LAST WORD IS A VERB
       TAKING STATEMENT OBJECT > #CSTATEMENT } /
       { !THAT #CSTATEMENT } /
       { { - { !, } #PREPPHR
       * { CONJ #PREPPHR } } /
       ADV /
       ADJ /
#PPCL /
#VINGCL /
#VERB-COMPLEX-ARGUMENT /
{ !, #INTERRUPTER !, } /
{ !{ #INTERRUPTER !} } } }

NET-DEF #GINF
  { * { ADV }
   { { INF #VCOMP } /
      { !HAVE VERBPP #VCOMP } /
      { !HAVE !BEEN { VERBPP / VERBING } #VCOMP  } /
        { !BE * { ADV } { VERBPP / VERBING } #VCOMP
     * { CONJ { VERBPP / VERBING } #VCOMP } } } }
```

NOUN PHRASES

```
NET-DEF #CNP
  { - { !BOTH / !EITHER / !NEITHER }
    { #NP
       * { + { !, / !; / CONJ } #NP }
       - { !, } } }

NET-DEF #NP
  { { RELPRN * { + { CONJ / !, } RELPRN }
     #DECLARATIVE-STATEMENT } /
   { RELPRN * { + { CONJ / !, } RELPRN }
     !TO #GINF #VCOMP } /
     { !WHETHER #DECLARATIVE-STATEMENT
      !OR  { !NOT / #DECLARATIVE-STATEMENT } }
   { * { PREDETADJ }
      - { DEFDET / NDEFDET }
      * { NOUN / { ADJ - { CONJ } } }
      NOUN
      - { #RELCLAUSE /
          { POSS-MARK #CNP } /
          { !{ #CNP !} } } } /
      NAME /
      PROPPRN /
      PRN /
      #GERUND }

NET-DEF #GERUND
  { - { NEG } { VERBING / !HAVING VERBPP } #VCOMP }

NET-DEF #PREPPHR
  { PREP #CNP
    * { + { CONJ / !, } PREP #CNP } }

NET-DEF #RELCLAUSE
  { #FOR-TO-RELCLAUSE / #SUBRELCLS / #OBJRELCLS }

NET-DEF #INTERRUPTER
  { #COMPOUND-MODIFIER /
   #CNP /
   #PPCL }
```